arXiv:2006.09503v3 [cs.LG] 22 Jul 2021

# Memory-Efficient Pipeline-Parallel DNN Training

**Deepak Narayanan** [1 *] **Amar Phanishayee** [2] **Kaiyu Shi** [3] **Xie Chen** [3] **Matei Zaharia** [1]

## Abstract

Many state-of-the-art ML results have been obtained by scaling up the number of parameters in existing models. However, parameters and activations for such large models often do not fit in the memory of a single accelerator device; this means that it is necessary to distribute training of large models over multiple accelerators. In this work, we propose PipeDream-2BW, a system that supports *memory-efficient* pipeline parallelism. PipeDream-2BW uses a novel pipelining and weight gradient coalescing strategy, combined with the double buffering of weights, to ensure high throughput, low memory footprint, and weight update semantics similar to data parallelism. In addition, PipeDream-2BW automatically partitions the model over the available hardware resources, while respecting hardware constraints such as memory capacities of accelerators and interconnect topologies. PipeDream-2BW can accelerate the training of large GPT and BERT language models by up to 20× with similar final model accuracy.

## 1. Introduction

In the quest to achieve higher accuracy across a range of tasks, DNN models have grown in size, often by scaling up the number of parameters in existing architectures (Devlin et al., 2018; Radford et al., 2018; 2019; Brown et al., 2020). It is challenging to train large models with billions of parameters. Modern accelerators have limited memory, which means that the model parameters and intermediate outputs that need to be in accelerator memory during training might not fit on a single accelerator. One of the solutions researchers and practitioners have turned to is model-parallel training (Dean et al., 2012; Chilimbi et al., 2014), where a model is partitioned over multiple accelerator devices. However, model parallelism, when traditionally deployed, can either lead to resource under-utilization (Narayanan et al., 2019) or high communication overhead with good scaling only within a multi-GPU server (Shoeybi et al., 2019), and consequently an increase in training time and dollar cost.

Recent work has proposed *pipelined* model parallelism to accelerate model-parallel training. For example, GPipe (Huang et al., 2019) and PipeDream (Harlap et al., 2018; Narayanan et al., 2019) push multiple inputs in sequence through a series of workers that each manage one model partition, allowing different workers to process different inputs in parallel. Naïve pipelining can harm model convergence due to inconsistent weight versions between the forward and backward passes of a particular input. Existing techniques trade off memory footprint and throughput in different ways to avoid this. GPipe maintains a single weight version, but has periodic *pipeline flushes* where the pipeline is drained of inputs to update weights (Figure 1a); these flushes limit overall throughput as resources are idle. PipeDream does not periodically flush the pipeline but stores multiple weight versions, which increases throughput but also increases the memory footprint, making the training of large models infeasible due to memory constraints. Efficient training of large models requires an approach with *both* high throughput and low memory footprint.

Additionally, the performance of a pipeline-parallel system is dependent on how DNN model operators are partitioned over workers. This is challenging for three reasons:

- **Memory Capacity Constraints:** Parameters and intermediate activations associated with a model partition need to fit in the main device memory of the accelerator.
- **Heterogeneous Network Interconnects:** Training deployments today feature heterogeneous network topologies, with higher-bandwidth links between devices on the same server.
- **Large Search Space for Operator Placement:** As model sizes increase, splitting an operator graph becomes computationally expensive since the number of distinct partitionings is exponential in the model size.

In this paper, we introduce **PipeDream-2BW**, a system for efficient pipeline-parallel training of DNN models with billions of parameters. PipeDream-2BW achieves high through-

[*]Work done in part as intern at Microsoft Research. [1]Stanford University [2]Microsoft Research [3]Microsoft. Correspondence to: Deepak Narayanan <deepakn@cs.stanford.edu>.

*Proceedings of the $38^{th}$ International Conference on Machine Learning*, PMLR 139, 2021. Copyright 2021 by the author(s).

put *and* low memory footprint using two key contributions. First, we propose **double-buffered weight updates** (2BW), a technique that reduces the memory footprint of training while avoiding pipeline flushes. We leverage the fact that every input's generated gradient does not need to be applied to weights immediately, and instead can be accumulated into a "coalesced" gradient to limit the number of weight versions maintained. Instead of flushing the pipeline before using newly updated weights, 2BW uses the new weights for inputs newly admitted into the pipeline, while using the previous weight version, called the *shadow version*, for already in-flight inputs. This double buffering of weights at each worker yields a pipelining scheme with higher throughput than GPipe (no pipeline flushes) and better memory efficiency than PipeDream (2 weight versions, versus worst case of $d$ in PipeDream for a depth-$d$ pipeline). 2BW introduces a *constant* weight delay term of 1, consistent across stages, while updating weights (weight update equation of $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t-1)})$), which we show has empirically similar model convergence to vanilla weight updates (§5.1). We also present a variant of 2BW (called **PipeDream-Flush**) that trades off throughput for even lower memory footprint and vanilla semantics (weight update equation of $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$).

Second, PipeDream-2BW provides a planning algorithm that yields effective parallelization schemes for many of today's large model architectures. PipeDream-2BW's **planner** partitions DNN operators over the available workers while taking into account the memory capacities of the accelerator devices, and addresses the three challenges highlighted earlier. PipeDream-2BW's planner exploits the repetitive structure of large DNNs, e.g., transformer layers in BERT (Devlin et al., 2018), to explore the space of schedules where each stage in the pipeline is replicated *equally*. This choice reduces the size of the search space explored drastically compared to existing work like PipeDream and FlexFlow (Jia et al., 2018), while still providing effective model splits in practice. PipeDream-2BW's planner determines the size of each model partition, batch size, and whether to use memory-saving optimizations like *activation recomputation* (Chen et al., 2016; Griewank & Walther, 2000). PipeDream-2BW's planner considers the impact of these decisions on both throughput and memory footprint, unlike PipeDream and FlexFlow. Finally, the planner tries to ensure expensive communication stays on high-speed intra-server interconnects.

We find that the Adam optimizer with 2BW has a similar training loss trajectory to vanilla Adam with the same batch size, with similar accuracy on downstream finetuning tasks. PipeDream-2BW achieves end-to-end speedups of 1.3× to 20× for various GPT models compared to an optimized model-parallel baseline. PipeDream-2BW is up to 3.2× faster than GPipe, and is able to train large transformer models that vanilla PipeDream cannot fit in memory.

## 2. Background

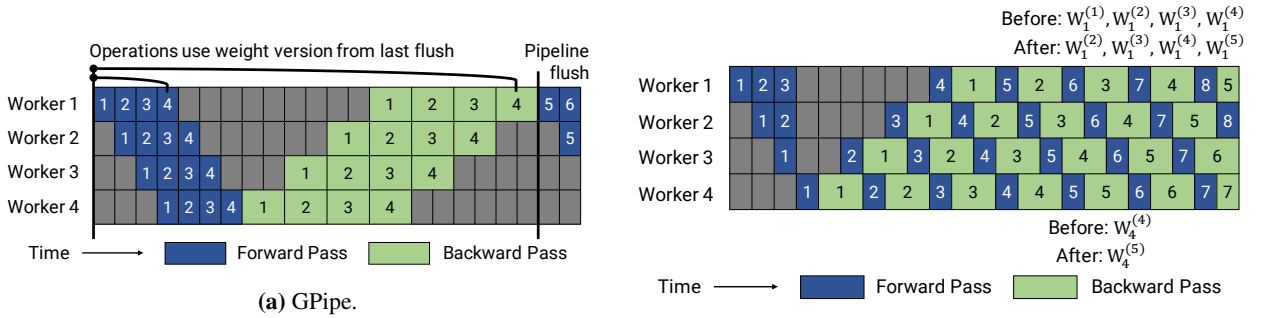
In this section, we provide a brief overview of related techniques for distributed training of DNN models.

**Data Parallelism.** Data parallelism is used to scale up model training. With data parallelism (Xing et al., 2015), every worker has a copy of the entire model and the input dataset is sharded across workers. Data parallelism cannot be used to train large models that do not fit on a single worker, but can be used on smaller model partitions.

**Model Parallelism.** Model parallelism is used traditionally to train large models that do not fit on a single worker. With model parallelism (Dean et al., 2012; Chilimbi et al., 2014), the weight parameters in a model are split over available workers, with intermediate activations and gradients communicated across workers. Inter-layer model parallelism underutilizes resources since at most a single worker is active at any point in time. Tensor (intra-layer) model parallelism (Shoeybi et al., 2019) leads to expensive all-to-all communication in the critical path, limiting the number of model partitions to the number of GPUs in a single server. FlexFlow (Jia et al., 2018) shows how to split a model graph using model and data parallelism, but still suffers from poor resource utilization when model parallelism is used.

**Pipeline Parallelism.** To address the shortcomings of model parallelism, recent work like PipeDream and GPipe have proposed pipeline parallelism. With pipeline parallelism, multiple inputs (instead of 1) are injected into a pipeline composed of inter-layer model partitions. This ensures that compute resources are better utilized. However, naive pipelining can lead to weight version mismatches between forward and backward passes for a particular input. Specifically, if weight updates are immediately applied to the latest weight version, then an input might see weight updates in the backward pass that it did not see in the forward pass, leading to incorrect gradient computations.

GPipe maintains a single version of the model's weights. An input batch is split into smaller *microbatches*. Weight gradients are accumulated and not applied immediately, and the pipeline is periodically *flushed* to ensure that multiple weight versions do not need to be maintained. GPipe provides weight update semantics similar to data parallelism. Figure 1a shows a timeline of GPipe execution. The periodic pipeline flushes can be expensive, limiting throughput. One way to mitigate this overhead is to perform additional accumulation within the pipeline, but this is not always practical: a) at large scale factors, the minimum supported batch size is large (proportional to the scale factor), and large batch sizes affect convergence for all models (e.g., Megatron (Shoeybi et al., 2019) uses a batch size of 1024 for BERT and 512 for GPT with 512 GPUs), b) GPipe needs to maintain activation stashes proportional to the batch size.




**(a)** GPipe.

**(b)** PipeDream.

*Figure 1.* Timelines of different pipeline-parallel executions. Without loss of generality, forward and backward passes are assumed to take twice as long as forward passes; forward passes are shown in blue and backward passes are shown in green. Numbers indicate microbatch ID, time is shown along $x$-axis, per-worker utilization is shown along the $y$-axis. GPipe maintains a single weight version, but periodically flushes the pipeline. PipeDream does not introduce periodic pipeline flushes, but maintains multiple weight versions.

PipeDream uses a weight stashing scheme to ensure that the same weight version is used in both the forward and backward passes for the same input (Figure 1b). The total number of weight versions stashed is $d$ in the worst case, where $d$ is the pipeline depth, which is too high for large models. With PipeDream's default weight update semantics, weight updates at each stage have different delay terms, and no accumulation is performed within the pipeline.

## 3. PipeDream-2BW System Design

PipeDream-2BW uses memory-efficient pipeline parallelism to train large models that do not fit on a single accelerator. Its *double-buffered weight update (2BW)* and *flush* mechanisms ensure high throughput, low memory footprint, and weight update semantics similar to data parallelism. PipeDream-2BW splits models into stages over multiple workers, and replicates each stage an equal number of times (with data-parallel updates across replicas of the same stage). Such *parallel pipelines* work well for models where each layer is repeated a fixed number of times (e.g., transformer models).

### 3.1. Double-Buffered Weight Updates (2BW)

PipeDream-2BW uses a novel double-buffered weight update (2BW) scheme in conjunction with 1F1B scheduling (Narayanan et al., 2019), where each worker alternates between forward and backward passes for different inputs, to ensure that the same weight version is used in both the forward and the backward pass for a particular input (Figure 2). 2BW has a lower memory footprint than PipeDream and GPipe, and also avoids GPipe's expensive pipeline flushes.

Gradients are computed at the granularity of smaller *microbatches*. For any input microbatch, PipeDream-2BW uses the same weight version for an input's forward and backward passes. Updates are accumulated over multiple microbatches before being applied at the granularity of a batch, limiting the number of weight versions generated and maintained. Figure 2 shows an example timeline of 2BW. PipeDream-2BW generates a new weight version once every $m$ microbatches ($m \geq d$, the pipeline depth). For simplicity, we will initially assume that $m = d$ ($d = 4$ in Figure 2). A new weight version *cannot* be used immediately. In particular, in-flight inputs cannot use the newest weight version for their backward passes (for example, input 7 on worker 3 at $t = 21$), since the forward pass for these inputs was already initiated using an older weight version on a different stage. Thus, newly generated weight versions need to be buffered for future use. However, the total number of weight versions that need to be maintained is at most 2, since the weight version used to generate a new weight version can immediately be discarded (no future inputs that pass through that stage use the old weight version any longer). For example, in Figure 2, each worker can discard $W_i^{(0)}$ once they are done processing the *backward pass* for input 8 since all subsequent inputs use a later weight version for both their forward and backward passes.

The weight version a given input microbatch $k$ (1-indexed) uses is $\max(\lfloor (k-1)/m \rfloor - 1, 0)$, where $m$ is the number of microbatches in a batch (4 in Figure 2). This weight version is the same for both the forward and backward passes for input $k$. $m$ can be any number $\geq d$; additional gradient accumulation (larger $m$) increases the global batch size.

**Memory Footprint.** PipeDream-2BW maintains 2 weight versions, and activation stashes for all in-flight microbatches. The number of in-flight microbatches at any stage is at most the pipeline depth ($d$). With activation recomputation, PipeDream-2BW's memory footprint can be decreased, since only input activations (as opposed to the full intermediate activation) need to be maintained for all in-flight microbatches. With activation recomputation, PipeDream-2BW's worst-case memory footprint is $\frac{2|W|}{d} + \frac{|A^{\text{total}}(b)|}{d} + d|A^{\text{input}}(b)|$. $|W|$ is the size of weight parameters for the full model, $|A^{\text{total}}(b)|$ is the size of intermediate activations for microbatch size $b$ for the full model, and $|A^{\text{input}}(b)|$ is the size of

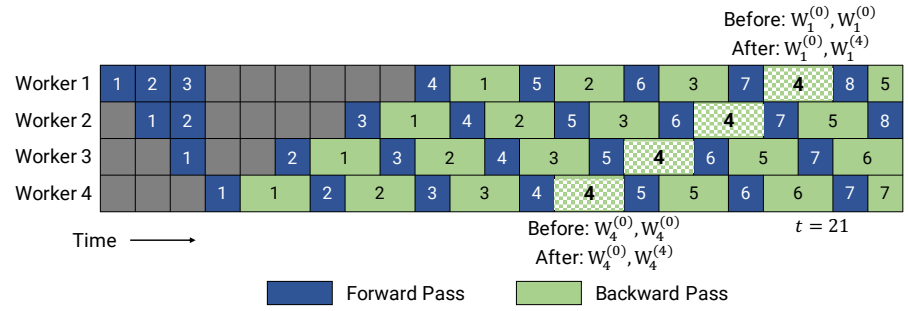
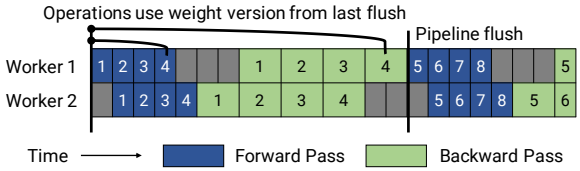



*Figure 2.* Timeline showing PipeDream-2BW's double-buffered weight update (2BW) scheme with time along $x$-axis. *Without loss of generality*, backward passes are assumed to take twice as long as forward passes. PipeDream-2BW only stashes two weight versions at every worker, reducing the total memory footprint while no longer requiring expensive pipeline stalls. $W_i^{(v)}$ indicates weights on worker $i$ with version $v$ (contains weight gradient generated from input $v$). New weight versions are generated in checkered green boxes; $W_4^{(4)}$ is first used for input 9's forward pass.




**(a)** GPipe.


**(b)** PipeDream-Flush.

*Figure 3.* Timelines of GPipe and PipeDream-Flush for 2 stages. Both GPipe and PipeDream-Flush use pipeline flushes; PipeDream-Flush alternates between forward and backward passes in steady state to keeping memory footprint low compared to GPipe by limiting activation stashes to only in-flight microbatches.

input activations for microbatch size $b$ *for a pipeline stage*.

In comparison, GPipe needs to checkpoint potentially a much larger number of input activations – proportional to the total number of microbatches accumulated within the pipeline before applying a weight update ($m$). With activation recomputation, GPipe's memory footprint with a per-GPU microbatch size $b$ is $\frac{|W|}{d} + \frac{|A^{\text{total}}(b)|}{d} + m|A^{\text{input}}(b)|$. Since $|W| \ll |A(b)|$ for even small $b$ for most models (Jain et al., 2018), the memory savings from maintaining one fewer weight version is small. To achieve high throughput, GPipe must use a large value of $m$ to amortize away the cost of pipeline flushes; at such high $m$, its memory footprint is higher than PipeDream-2BW. Additionally, due to its higher memory footprint, GPipe must always use activation recomputation. Activation recomputation, however, reduces throughput by about 33%, and should be avoided if possible.

**Semantics.** We can also formalize the semantics of 2BW.

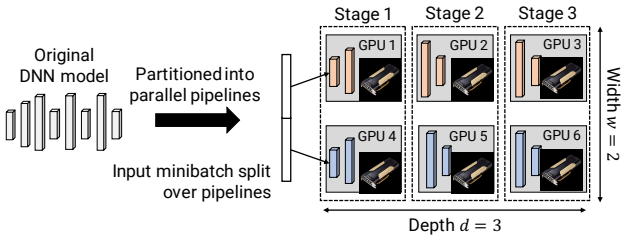



*Figure 4.* Example PipeDream-2BW $(2, 3)$ configuration. The model is partitioned into 3 stages ($d = 3$) and each pipeline is replicated twice ($w = 2$). Each pipeline replica is shown in a different color.

For this discussion, we assume an unreplicated pipeline with $d$ stages. If $b$ is the per-GPU microbatch size, then gradients are averaged over $m$ microbatches; thus, the effective batch size is $B = b \cdot m$.

We denote $W^{(t)}$ as the weight version after $t$ batches of size $B$. $\nabla f(W)$ is the gradient averaged over the $B$ samples in the batch. Vanilla minibatch SGD ($f$ is the loss function, $\nu$ is the learning rate) then has the following weight update equation: $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$. 2BW's weight update semantics (with a delay term of 1 *across* all stages) are almost unchanged:

$$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t-1)}).$$

We show that this delay term does not affect model convergence significantly in §5.1. Intuitively, the parameters of the model do not change significantly across single iterations, so $W^{(t)} \approx W^{(t-1)}$. The semantics with a replication factor greater than 1 is similar, with the batch size multiplied by the number of replicas (as with regular data parallelism). Other momentum-based optimizers such as Adam can be similarly analyzed (momentum term uses a weight gradient computed on a 1-stale weight version instead of latest version). Extra shadow variables are not needed. For example, $m_t$ in minibatch SGD with momentum can be computed as (ignoring bias corrections)

$m_t = \beta \cdot m_{t-1} + (1 - \beta) \cdot \nabla f(W^{(t-1)})$. The final weight update equation is then $W^{(t+1)} = W^{(t)} - \nu \cdot m_t$.

### 3.2. Weight Updates with Flushes (PipeDream-Flush)

We also propose a second memory-efficient pipeline schedule called PipeDream-Flush. It has lower memory footprint than 2BW and vanilla optimizer semantics, at the cost of lower throughput. This schedule reuses the 1F1B schedule from PipeDream (Narayanan et al., 2019), but maintains a single weight version and introduces periodic pipeline flushes to ensure consistent weight versions across weight updates. Timelines for PipeDream-Flush and GPipe with 2 pipeline stages are shown in Figure 3.

**Memory Footprint.** With PipeDream-Flush, the total number of in-flight "active" input activations is less than or equal to the pipeline depth, giving it lower memory footprint than GPipe, which has to maintain input activations proportional to the number of microbatches over which gradients are averaged ($m$). PipeDream-Flush's memory footprint is also lower than PipeDream-2BW since it only needs to maintain a single weight version (versus 2 with PipeDream-2BW).

**Semantics.** Periodic pipeline flushes ensure that weight updates can be performed with gradients computed using the latest weight version. This results in weight updates of the form $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$. We compare 2BW's statistical efficiency (rate of model convergence) to the vanilla semantics of PipeDream-Flush, GPipe, and data parallelism, in §5.1.

### 3.3. Equi-replicated Stages (Parallel Pipelines)

PipeDream-2BW executes DNN training using a hybrid parallelization scheme which combines data and model parallelism with input pipelining. Since large deep models today feature extremely repetitive structures, with the same block repeated multiple times, a simple way of load balancing computation and communication involves breaking up a model into stages with an equal number of blocks and replication factors. Model training in PipeDream-2BW can thus be thought of as a collection of parallel pipelines (Figure 4), where inputs and intermediate output activations within a pipeline do not ever need to be sent to workers responsible for a different pipeline. Intermediate activations and gradients can be communicated *within* a pipeline using point-to-point communication primitives, such as `send` and `recv`. As with PipeDream, weight gradients need to be aggregated across stage replicas in different pipelines. Figure 4 shows an example: each model copy is split across 3 workers (number of stages or depth, $d = 3$), and each stage is replicated twice (number of pipelines or width, $w = 2$). Stage replicas can be placed on the same server so that expensive all-reduce updates are between GPUs on the same server with high-bandwidth interconnects.

## 4. Planner

PipeDream-2BW's *planner* determines how to split a model over the available compute devices by exhaustively searching over the *reduced* search space of all possible parallel-pipeline configurations. The planner also determines whether memory-saving optimizations should be deployed, and the per-GPU microbatch size and degree of gradient accumulation, given a maximum *safe* global batch size verified to not compromise model convergence (e.g., determined from past hyperparameter sweeps without pipelining).

PipeDream-2BW's planner uses a cost model for the compute times and memory footprints of individual blocks in the model. Time and memory cost functions allow PipeDream-2BW to reason about the impact of pipeline width / depth and memory-saving optimizations (such as activation recomputation) on throughput and memory footprint. For example, a deeper configuration has additional memory capacity, allowing for a larger maximum per-GPU microbatch size; this can increase the arithmetic intensity (number of floating point operations performed per memory load) of kernels (Jouppi et al., 2017), and consequently throughput. Communication times for tensors can be estimated by dividing the size of the tensor by the respective bandwidth. Expensive communication (e.g., large tensors, or all-reduce communication needed to coalesce weight gradients across stage replicas) can be placed on high-bandwidth links within the server by orienting pipelines appropriately.

Profiling for cost modeling can be done in two ways: end-to-end for each distinct configuration, or extrapolating from an individual block's measurements. End-to-end profiling is cheap (2 to 3 minutes per configuration), which means total profiling time is still a couple of hours (compared to the days to weeks needed for model training). Optimal configurations can be reused for a given server and model deployment. We describe how per-block time and memory measurements can be extrapolated in Appendix §A – this is even cheaper, but provides less accurate cost estimates. The highest-throughput-configuration is chosen that also fits within the memory capacity of the target accelerators.

### 4.1. Activation Recomputation

Activation recomputation is a common technique (Huang et al., 2019; Chen et al., 2016; Griewank & Walther, 2000) that trades off extra computation for a lower memory footprint. With activation recomputation, activation stashes are not left materialized on the device between forward and backward passes; instead, only *input* activations on each stage are stashed, and the remaining activations needed in the backward pass are recomputed when required by re-running the forward pass. Activation recomputation trades off extra computation for a lower memory footprint.

Activation recomputation is useful for two reasons: it can enable larger per-GPU microbatch sizes to fit in memory, which can improve device throughput by increasing the arithmetic intensity of kernel. It can also enable the training of large models. Concretely, in some cases, the target accelerator device does not have sufficient memory capacity to store full activation stashes for all in-flight microbatches. This is especially true for deep pipelines, since the number of in-flight inputs is proportional to the depth of the pipeline (Narayanan et al., 2019).

### 4.2. Partitioning Algorithm

Putting it all together, given a total memory capacity $M$, PipeDream-2BW's planner first determines the largest per-GPU microbatch size that fits on a given worker (and the corresponding throughput) with and without each memory-savings optimization deployed using a memory cost function. The partitioning algorithm also verifies that the resulting global batch size is lower than the maximum safe batch size $B$ . Each memory-savings optimization can be integrated into PipeDream-2BW's planner by specifying a corresponding throughput and memory cost function.

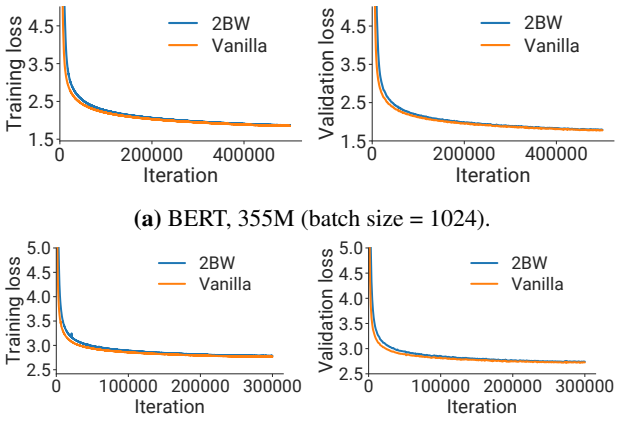
PipeDream-2BW's planner then sweeps all $(w, d)$ values to determine the best pipeline configuration for a given model and hardware deployment. Configurations with memory footprint higher than the memory capacity $M$ of the device (modeled by the MEMORY(.) cost function) are discarded. Gradient accumulation can be used to increase the batch size to $B$. The partitioning algorithm aims to pick a configuration that has a high compute-to-communication ratio, while accounting for the communication time across stages in the same pipeline and across replicated stages (modeled by the THROUGHPUT(.) cost function). The full algorithm is shown in Appendix §A.

## 5. Evaluation

In this section, we show that the Adam optimizer with 2BW has similar semantics to vanilla Adam, and that PipeDream-2BW and PipeDream-Flush are able to train large models faster than existing model-parallel approaches including Megatron (Shoeybi et al., 2019), and existing pipelining approaches like GPipe (Huang et al., 2019).

**Hardware.** We show results on two different hardware setups on AWS: eight 8×V100 servers (64 GPUs) with NVLink and 16GB of per-GPU memory, and a single 8×V100 server. We use p3.16xlarge instances.

**Implementation.** Our implementation uses PyTorch and is adapted from the Megatron repository (meg); we verified that single-worker performance with this implementation achieves about 45 TFLOPS on a 355M-parameter GPT model and is competitive with existing state-of-the-art open source implementations from NVIDIA (nvi). All results shown are with mixed precision.




**(a)** BERT, 355M (batch size = 1024).

**(b)** GPT, 355M (batch size = 512).

*Figure 5.* Training and validation loss when pre-training BERT and GPT models with vanilla Adam and Adam with 2BW.

**Models.** We evaluate PipeDream-2BW on BERT (Devlin et al., 2018) and GPT (Radford et al., 2019), large transformer-based language models used for a number of NLP applications. In particular, most of our experiments are performed with GPT models with 1.3, 2.2, and 3.9 billion parameters, with similar layer dimensions to those used in the Megatron paper (Shoeybi et al., 2019).

**Baselines.** We compare PipeDream-2BW to two types of baselines: (a) model parallelism without pipelining (tensor model parallelism used in Megatron, and inter-layer model parallelism); and (b) GPipe (we extend GPipe to use parallel pipelines, and refer to this *enhanced* version as GPipe in the rest of this paper), which performs pipeline parallelism. We do not compare to PipeDream or data parallelism for the entire model since they cannot fit the above models in memory when using 16-GB V100 GPUs. With 64 GPUs, we use data parallelism *across stages* to scale up training.

**Main Takeaways.** We make the following observations:

- **Quality of Convergence:** 2BW weight update semantics yield pre-trained models which produce **comparable accuracy** on downstream finetuning tasks to vanilla Adam (GPipe and PipeDream-Flush) with the same batch size.
- **Comparison to Model Parallelism:** PipeDream-2BW is able to train a 3.8 billion-parameter GPT model up to **20**× faster compared to non-pipelining approaches.
- **Comparison to Other Pipelined Approaches:** PipeDream-2BW is up to **3.2**× faster than GPipe.

### 5.1. Quality of Convergence of 2BW

We pre-trained 355M-parameter BERT and GPT models with vanilla Adam and Adam with 2BW; we then finetuned

| Task | Metric | Vanilla | Vanilla (90%) | 2BW |
|---|---|---|---|---|
| MNLI | Overall Acc. | 87.77% | N/A | 87.82% |
| RACE | Overall Acc. | 80.06% | 79.30% | 79.48% |

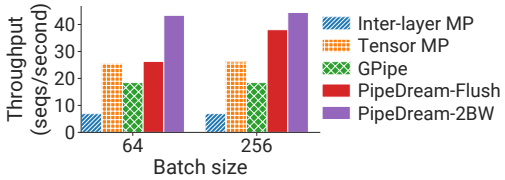
*Table 1.* Comparison of BERT models pre-trained with vanilla (all and 90% of iterations) and 2BW optimizers on finetuning tasks.

the resulting BERT models. We note that GPipe, PipeDream-Flush, and DP have identical semantics, and hence are equivalent baselines ("Vanilla"). To provide a fair comparison, we use the *same* hyperparameters, including batch size, used by Megatron (Shoeybi et al., 2019) to train these BERT and GPT models. For BERT, we use a batch size of 1024, and for GPT, we use a batch size of 512. We use the Adam optimizer with standard hyperparameters (learning rate of $10^{-4}$ with initial warmup and subsequent linear decay, maximum sequence length of 512), and mixed precision. We used the OpenWebText dataset (ope) for pretraining. Figure 5 shows the training and validation loss for the two models. The training and validation losses for the 2BW runs track the vanilla runs almost identically after the first 100k iterations (when the model is changing more rapidly and the delay term matters more).

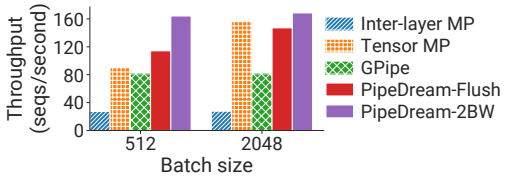
To further validate the quality of the pre-trained model, we finetuned the pre-trained vanilla and 2BW BERT models on downstream MNLI and RACE tasks (Wang et al., 2019; Lai et al., 2017). Both pre-training and fine-tuning were performed with the same hyperparameter and training setups, and we did not perform hyperparameter tuning for either – our goal here is to show that 2BW has nearly identical semantics to the corresponding vanilla optimizer. As shown in Table 1, the accuracy on each of these tasks is similar after finetuning. We also evaluated the vanilla and 2BW GPT models on the Wikitext-103 test dataset and got similar test perplexities (19.28 vs. 19.56); test perplexities match *exactly* when "Vanilla" is run for 20% fewer iterations.

### 5.2. Throughput

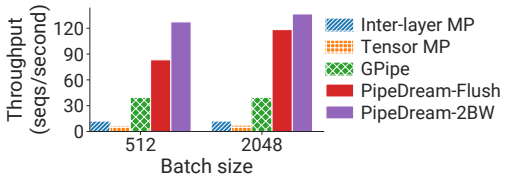
Figure 6 shows the throughputs of various PipeDream-2BW, PipeDream-Flush, and baseline configurations using 8 and 64 V100s with a sequence length of 512 for various large GPT models. Results with BERT models are similar and included in Appendix §B.1. We compare to two different forms of model parallelism, as well as GPipe. Data parallelism is not a viable baseline for these large models due to its high memory overhead. In these experiments, we use activation recomputation, and the largest per-GPU microbatch size that fits on the 16-GB V100 GPUs. We use the best configuration recommended by PipeDream-2BW's planner for all comparisons: 8-deep configurations for the model with 2.2 billion parameters, and 16-deep configurations for the model with 3.8 billion parameters. For each model, we show two different batch sizes to show the impact of batch




**(a)** GPT, 2.2B, 8-way model parallelism (8×V100s).




**(b)** GPT, 2.2B, 8-way model parallelism (64×V100s).




**(c)** GPT, 3.8B, 16-way model parallelism (64×V100s).

*Figure 6.* Throughput of various systems for different batch sizes for GPT models, using 8×16GB-V100 servers.

size on throughput for approaches that use periodic flushes.

**Model Parallelism without Pipelining:** We compare against two model parallelism approaches: tensor model parallelism used by Megatron (Shoeybi et al., 2019) where each layer is divided among all model-parallel workers, and inter-layer model parallelism where layers are sharded over the workers but inputs are not pipelined. On a single node, PipeDream-2BW is faster than tensor MP by **1.3**×. This grows to **20**× on 64 GPUs for the model with 3.8 billion parameters, when the all-to-all communication used by tensor MP needs to be performed across servers, which is expensive using AWS instances (bandwidth across multi-GPU servers is much lower than the bandwidth within server). Compared to inter-layer MP, pipelining with flushes increases throughput by up to **4.1**× for small batch sizes, and by up to **5.3**× for large batch sizes, on the 2.2-billion model. 2BW is up to **6.1**× faster than inter-layer MP.

**GPipe:** PipeDream-2BW outperforms corresponding GPipe configurations at the same global batch size by up to **3.2**× due to the lack of periodic pipeline flushes. GPipe natively has high memory footprint due to a large number of activation stashes: consequently, the maximum number of microbatches it can admit is small, leading to a larger pipeline bubble and **2.1**× worse throughput than PipeDream-Flush at low batch sizes, and **3**× at high batch sizes.

**PipeDream-Flush and PipeDream-2BW:** Figure 6 also compares PipeDream-2BW and PipeDream-Flush for two different batch sizes with different numbers of microbatches over which gradients are averaged ($m = d \cdot g$) within the pipeline. At low batch size, PipeDream-2BW is up to **1.6**×

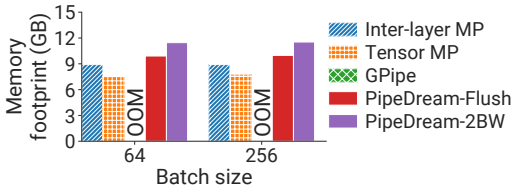



*Figure 7.* Worst-case memory footprint (in GB) of various systems with 8 V100 GPUs for a GPT model with 2.2 billion parameters.

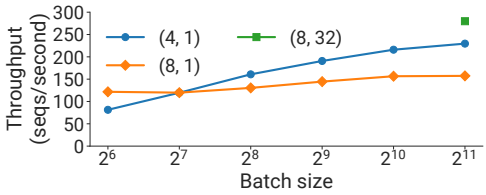



*Figure 8.* Throughput of two PipeDream-2BW configurations vs. global batch size for a 1.3-billion parameter GPT model using 64 V100 GPUs. The legend shows $(d, b)$: the number of pipeline-parallel stages and the microbatch size.

faster. With more gradient accumulation (batch size of 2048), this speedup drops to **15%**. However, high $g$ is not always practical. Both PipeDream-Flush and PipeDream-2BW have weight updates with a batch size of $b \cdot w \cdot d \cdot g$, where the total number of workers is $w \cdot d$. For a large number of workers ($\gg 64$), the batch size is high even with $g = 1, m = d$, making additional gradient accumulation infeasible (batch size cannot scale to $\infty$ without affecting model convergence). Indeed, systems like Megatron (Shoeybi et al., 2019), that train large transformer models using 512 GPUs, show state-of-the-art results across tasks using a global batch size $\leq 1024$.

### 5.3. Memory Footprint

We measured the worst-case memory footprint of different systems on a GPT model, shown in Figure 7. GPipe runs out of memory at a batch size of 64, due to a larger number of activation stashes from its all-forward-all-backward schedule, even with activation recomputation (worst case of $m$ input activation stashes with activation recomputation, compared to $d$ for PipeDream-Flush). PipeDream-Flush has a slightly higher memory footprint compared to inter-layer model parallelism, since it needs to maintain activation stashes for more in-flight microbatches. PipeDream-2BW has a higher memory footprint than PipeDream-Flush due to an additional weight version (but still lower than GPipe's).

### 5.4. Planning Decisions

In this sub-section, we analyze the implications of pipeline depth and width on performance. We show experiments demonstrating the impact of activation recomputation on performance in Appendix §B.2. Figure 8 shows the throughputs of two PipeDream-2BW configurations for different batch sizes. We highlight relevant takeaways below.

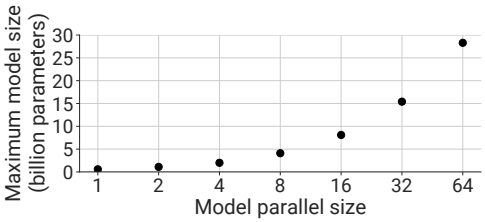



*Figure 9.* Maximum model size supported by various pipeline-parallel depths with 64 16-GB V100 GPUs.

**Inter-Stage Communication:** As the global batch size increases with gradient accumulation, throughput for each configuration increases due to less communication across stage replicas. This is especially true for configurations with communication across servers ($w > 8, d < 8$ for 8-GPU servers, e.g. $d = 4$) where inter-stage all-to-all communication is cross-node and more expensive.

**Compute-Communication Ratio:** Increasing the pipeline depth decreases the amount of computation in each pipeline stage while keeping the number of bytes communicated between stages constant. This makes the pipeline more communication-bound, decreasing throughput.

**Maximum Per-GPU Microbatch Size:** Increasing the pipeline depth increases the maximum microbatch size that fits in GPU memory. This leads to possibly higher arithmetic intensity and throughput. In Figure 8, we show throughput for two microbatch sizes for the $d = 8$ configuration; the larger microbatch size ($b = 32$) has higher throughput. Smaller pipeline depths cannot fit large microbatch sizes.

**Maximum Model Size:** Deeper pipelines support the training of larger models. We show the empirically measured maximum model size that can be trained with 2BW using different values of $d$ in Figure 9.

These observations illustrate the complexity in picking a configuration. For example, increasing pipeline depth leads to two effects (decreased compute-communication ratio within the pipeline and increased arithmetic intensity) that have opposing effects on throughput. PipeDream-2BW's planner automates this process for each combination of model, batch size, and number of GPUs.

### 5.5. Maximum Model Size Supported

Figure 9 shows the empirically measured maximum model size supported by various pipeline depths while using 2BW. As can be seen in the figure, deeper configurations provide additional memory capacity. PipeDream-2BW is able to train models of up to almost 30 billion parameters using 64 16-GB GPUs. As a point of comparison, Megatron-LM (Shoeybi et al., 2019) was able to train a model with 8.3 billion parameters with 8 32-GB GPUs (2× more memory).

## 6. Related Work and Discussion

In this section, we expand on work related to PipeDream-2BW, and place PipeDream-2BW's speedups in context.

**Model Parallelism in Real Deployments.** NVIDIA used a custom intra-layer model parallelism scheme in its Megatron system (Shoeybi et al., 2019) to train a GPT-2 model with 8.3 billion parameters on 64 32-GB V100 servers by parallelizing matrix multiplications across multiple workers. This approach can be combined with data parallelism. All-reductions are needed to coalesce partial results produced on different GPUs, thus making training communication-bound at high numbers of model partitions. In comparison, PipeDream-2BW trades off additional memory footprint (an extra weight version) for lower communication overhead (20× faster training when using multiple multi-GPU servers on Amazon AWS with limited inter-node bandwidth).

**Pipeline Parallelism.** We discussed the existing approaches to pipeline parallelism in §2, and showed quantitative comparisons in §5.2. PipeDream-2BW trains large models up to 3.2× faster than GPipe at low batch sizes, due to a lack of periodic pipeline flushes, and lower memory footprint that allows more input microbatches to be pushed into the pipeline. PipeDream cannot train these large models. PipeDream-2BW's lower memory footprint does come with tradeoffs, however – PipeDream-2BW accumulates weight gradients over multiple microbatches, increasing the minimum batch size that PipeDream-2BW supports. Thus, for models that only support very small batch sizes, PipeDream-2BW, PipeDream-Flush, and GPipe, which perform gradient accumulation within the pipeline, may not be viable.

PipeMare (Yang et al., 2019) uses asynchronous pipeline parallelism to provide high throughput (no pipeline flushes) with asynchronous weight update semantics. PipeMare offers two theoretically-motivated techniques to ensure good statistical efficiency. In contrast, PipeDream-2BW and all the baselines we compare against in the paper (traditional data parallel training, PipeDream, GPipe), use synchronous execution where the weights used for computation during forward propagation are the same as those used during backward propagation. PipeDream-2BW's double buffered weight updates use a 1-stale gradient update that does not require any hyperparameter tuning to generate comparable results. PipeMare also does not describe how computation should be partitioned among the available workers.

**Memory-Saving Optimizations.** A rich line of work attempts to decrease the memory footprint of DNN training. Gist (Jain et al., 2018) employs lossless and lossy layer-specific encoding schemes to compress stashed activations. Systems such as Checkmate (Jain et al., 2020) systematically determine when activation recomputation (Chen et al., 2016; Griewank & Walther, 2000) should be performed. DeepSpeed (Rajbhandari et al., 2019) partitions optimizer state over data-parallel replicas instead of replicating it, using a technique called ZeRO. Such orthogonal optimizations can be combined and incorporated in PipeDream-2BW.

**Planning Algorithms.** PipeDream, DAPPLE (Fan et al., 2021), and FlexFlow (Jia et al., 2018) use planning algorithms to partition operator graphs over multiple accelerators to maximize throughput. Unfortunately, these planners do not exploit the repetitive nature of modern transformer-based models. For example, PipeDream's planner explores $O(n^3m^2)$ configurations (assuming $n$ layers in the model and $m$ workers). Furthermore, these planners do not consider the effect of memory-saving optimizations, which are critical for training large models efficiently (e.g., always applying activation recomputation can make the system 1.33× slower). PipeDream-2BW's planner, on the other hand, performs an exhaustive search of a *much reduced* search space since it only considers *parallel pipelines* (all possible $(w, d)$ pairs with $m$ workers is $O(m^2)$). Given this small number of explored configurations, Bagpipe's planner takes a fraction of a second with a closed-form cost model; PipeDream's partitioning algorithm with the same cost model takes about 30 minutes for large models.

## 7. Conclusion

In this work, we proposed and implemented PipeDream-2BW, a system for memory-efficient pipeline-parallel training that achieves high throughput, low memory footprint, and data parallelism-like semantics through a novel weight update double buffering strategy called 2BW. PipeDream-2BW also uses a planner to determine how to partition a model's operator graph over training resources in a memory-aware way. PipeDream-2BW accelerates the training of models with billions of trainable parameters by up to 20× compared to model-parallel baselines, and by up to 3.2× compared to GPipe, on commodity hardware.

## Acknowledgements

We thank the anonymous reviewers, Aditya Grover, Paroma Varma, members of FutureData, and our colleagues at MSR for their feedback that improved this work. We thank MSR for their generous support of Deepak's internship, and for resources to develop and evaluate PipeDream-2BW. This research was also supported in part by affiliate members and other supporters of the Stanford DAWN project–Ant Financial, Facebook, Google, Infosys, NEC, and VMware–as well as Northrop Grumman, Amazon Web Services, Cisco, NSF Graduate Research Fellowship grant DGE-1656518, and the NSF CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors alone.

## References


Megatron Repository. https://github.com/nvidia/megatron-lm.

NVIDIA Deep Learning Examples, BERT. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/README.md#results.

OpenWebText Dataset. https://github.com/jcpeterson/openwebtext.

Brown, T., Mann, B., Ryder, N., Subbiah, M., and et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.

Chilimbi, T. M., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, volume 14, pp. 571–582, 2014.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 431–445, 2021.

Griewank, A. and Walther, A. Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv preprint arXiv:1806.03377*, 2018.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, pp. 103–112, 2019.

Jain, A., Phanishayee, A., Mars, J., Tang, L., and Pekhimenko, G. Gist: Efficient Data Encoding for Deep Neural Network Training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 776–789. IEEE, 2018.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pp. 497–511. 2020.

Jia, Z., Zaharia, M., and Aiken, A. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys)*, 2018.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

Lai, G., Xie, Q., Liu, H., Yang, Y., and Hovy, E. RACE: Large-scale ReAding Comprehension Dataset From Examinations. *arXiv preprint arXiv:1704.04683*, 2017.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving Language Understanding by Generative Pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 1(8):9, 2019.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*, 2019.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models using GPU Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. 2019. In the Proceedings of ICLR.

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., and Yu, Y. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C. R., and De Sa, C. PipeMare: Asynchronous Pipeline Parallel DNN Training. *arXiv preprint arXiv:1910.05124*, 2019.

# A. Planner, Additional Details

For every possible configuration of width and depth, PipeDream-2BW's planner explores the benefit of pipelining and each space-saving optimization. For example, with activation recomputation as a target memory-savings optimization, PipeDream-2BW considers three possible executions:

- Model and data parallelism without pipelining (with the largest per-GPU microbatch size that fits in memory).
- Hybrid parallelism with pipelining and without activation recomputation (all required weight versions and activation stashes in memory for in-flight microbatches).
- Hybrid parallelism with pipelining and recomputation.

PipeDream-2BW's planner estimates the throughput and memory footprint of each of these possible executions using a cost model. PipeDream-2BW's planner then tries to find the configuration with highest throughput that also fits in main device memory of the accelerators used (memory capacity provided as input). In this section, we show one such cost model for throughput and memory.

## A.1. Closed-Form Cost Functions

In our experiments, we used profile-based cost functions that run configurations end-to-end for a couple of hundred iterations. However, performance of different parallel configurations can also be estimated using closed-form expressions that use more fine-grained profile information (e.g., time and memory footprint of each transformer block). We present one such cost model here.

### A.1.1. THROUGHPUT(.) COST FUNCTION

The throughput of various hybrid-parallel setups with and without pipelining can be modeled using the times of forward and backward passes obtained from a simple profiling step. Let $b$ be the largest per-GPU microbatch size without additional weight and activation versions, and $b'$ be the largest per-GPU microbatch size that can fit on the device when multiple versions are needed ($b' \leq b$). As before, $w$ and $d$ are the pipeline width and depth.

Let $T_i^{\text{comp}}(b, w, d)$ represent the compute time of stage $i$ with a per-GPU microbatch size $b$, $T_{i \rightarrow j}^{\text{comm}}(b, w, d)$ represent the communication time of activations and gradients between stages $i$ and $j$ with microbatch size $b$, and $T_i^{\text{comm}}(b, w, d)$ represent the communication time of exchanging gradients between $w$ replicas of stage $i$ with microbatch size $b$. We assume that the global batch size used is $B$. With pipeline width $w$ and microbatch size $b$, data-parallel communication is required every $m(b) = B/(w \cdot b)$ microbatches.

Then, without pipelining, each microbatch of size $b$ takes the following computation time, $t$:

$$t = \sum_i \max(T_i^{\text{comp}}(b, w, d) + \sum_j T_{j \rightarrow i}^{\text{comm}}(b, w, d), \frac{1}{m(b)} \cdot T_i^{\text{comm}}(b, w, d))$$

With pipelining, computation of different stages can be overlapped. A microbatch of size $b'$ can then be processed every $t$ seconds, where $t$ is given by the expression:

$$t = \max_i \max(T_i^{\text{comp}}(b', w, d) + \sum_j T_{j \rightarrow i}^{\text{comm}}(b', w, d), \frac{1}{m(b')} \cdot T_i^{\text{comm}}(b', w, d))$$

With activation recomputation, the number of floating point operations increases, since forward passes need to be repeated to recompute the activation stashes needed in the backward pass. We use a constant multiplier $c^{\text{extra}}$ to represent this. $c^{\text{extra}} = 4/3$ is a reasonable value for this constant, since the backward pass typically takes twice as long as the forward pass. $c^{\text{extra}}$ can also be measured empirically. Arithmetic intensity might also increase, which is captured by $T_i^{\text{comp}}(.)$ being a function of the microbatch size $b$. Communication time remains unchanged from before. Every $b$ inputs can now be processed in time $t$, where $t$ is given by,

$$t = \max_i \max(c^{\text{extra}} \cdot T_i^{\text{comp}}(b, w, d) + \sum_j T_{j \rightarrow i}^{\text{comm}}(b, w, d), \frac{1}{m(b)} \cdot T_i^{\text{comm}}(b, w, d))$$

The throughput in samples per second of each of these setups is then the corresponding per-GPU microbatch size ($b$ or $b'$) divided by $t$.

**Estimating $T^{\text{comp}}(.)$.** $T_i^{\text{comp}}(b, w, d)$ is the compute time of stage $i$ with per-GPU microbatch size $b$, and can be computed by summing up the forward and backward pass times of all blocks within the stage. If the depth of the pipeline is $d$ and the total number of blocks in the model is $B$, then the total number of blocks in a given stage is $B/d$. Forward and backward pass times for each stage can be estimated by profiling 100–200 iterations of training.

**Estimating $T^{\text{comm}}(.)$.** Communication times can be similarly modeled. Let the size of the associated parameter with $B$ total blocks be $|W|$, and the size of the block's input and output activations be $|A^{\text{inp.+out.}}(b)|$. With a pipeline of depth $d$, each pipeline stage has $1/d$ of the total model parameters.

**Algorithm 1** Partitioning Algorithm

**Input:** Model $m$, memory capacity $M$, $m$'s associated search function SEARCH(.), $m$'s associated throughput cost function THROUGHPUT(.), $m$'s memory footprint cost function MEMORY(.), maximum safe batch size $B$.
**Return:** Optimal width and depth $w^{\text{opt}}$ and $d^{\text{opt}}$, optimal per-GPU microbatch size $b^{\text{opt}}$, boolean whether activations should be recomputed $r^{\text{opt}}$, optimal degree of gradient accumulation $g^{\text{opt}}$.

Initialize $t^{\text{max}} = 0, w^{\text{opt}} = \text{NULL}, d^{\text{opt}} = \text{NULL}$
**for** $w = 1$ **to** $N$ **do**
  **for** $d = 1$ **to** $N/w$ **do**
    *// For given width $w$, depth $d$, and batch size $B$, find optimal microbatch size and whether activation recomputation should be performed.*
    $b, r = m.\text{SEARCH}(w, d, B)$

    $t = m.\text{THROUGHPUT}(w, d, b, r)$
    **if** $m.\text{MEMORY}(w, d, b, r) > M$ **then**
      **continue**
    **end if**
    **if** $t > t^{\text{max}}$ **then**
      $t^{\text{max}} = t, w^{\text{opt}} = w, d^{\text{opt}} = d, b^{\text{opt}} = b, r^{\text{opt}} = r$
    **end if**
  **end for**
**end for**
$g^{\text{opt}} = B/(N \cdot b^{\text{opt}})$ *// To reach batch size $B$.*

The time to communicate activations across stages can be computed as (factor of 2 for gradients in the backward pass),

$$T^{\text{comm}}_{i \to j}(b, w, d) = \frac{2|A^{\text{inp.+out.}}(b)| \cdot \mathbb{I}(d > 1)}{\text{bwdth}_{\text{depth}}(d)}$$

The time to communicate weight gradients across stage replicas can be computed similarly given a bandwidth function $\text{bwdth}_{\text{width}}(w)$, and the number of bytes communicated during all-reduce. The number of byes communicated in an all-reduction can either be explicitly measured, or estimated using a closed-form expression (Narayanan et al., 2019).

$\text{bwdth}_{\text{depth}}(d)$ and $\text{bwdth}_{\text{width}}(w)$ represent the bandwidths for inter-stage and intra-stage communication. These bandwidth functions can respect hierarchical network topologies. For example, if $w$ is less than the number of workers in a single server, communication can be performed entirely within a server, using the higher intra-server bandwidth.

$$\text{bwdth}_{\text{width}}(w) = \begin{cases} B_{\text{high}} \text{ if } w < \text{number of GPUs in server} \\ B_{\text{low}} \text{ otherwise} \end{cases}$$

#### A.1.2. MEMORY(.) COST FUNCTION

The memory footprint can similarly be modeled using the sizes of activations and weights obtained from a profiling step. Let the total size of the weight parameters for the entire model be $|W|$, let the total size of the activations given a microbatch size $b$ for the entire model be $|A^{\text{total}}(b)|$, and let the size of the input activations for a single stage be $|A^{\text{input}}(b)|$. With a pipeline of $d$ stages, each pipeline stage has weight parameters of size $|W|/d$, and activations of size $|A^{\text{total}}(b)|/d$.

**Without Activation Recomputation.** As discussed in §3.1, 2BW maintains 2 different versions of the weight parameters. PipeDream-2BW also maintains $d$ versions of activations (the total number of in-flight activations). This means the total PipeDream-2BW memory footprint is:

$$\frac{2|W|}{d} + \frac{d|A^{\text{total}}(b)|}{d} + d|A^{\text{input}}(b)|.$$

**With Activation Recomputation.** With activation recomputation, the total number of activation versions in GPU memory at any point in time is 1. This means that the PipeDream-2BW memory footprint with $d$ stages is:

$$\frac{2|W|}{d} + \frac{|A^{\text{total}}(b)|}{d} + d|A^{\text{input}}(b)|.$$

### A.2. Partitioning Algorithm

We show pseudocode for the full partitioning algorithm in Algorithm 1.

## B. Evaluation, Additional Graphs

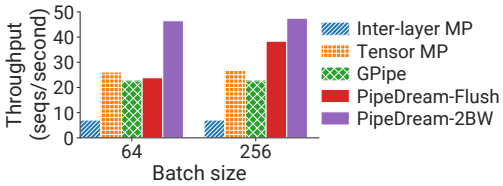
In this section, we present additional results we could not fit in the main paper due to space.

### B.1. Throughput and Memory Footprint with BERT Models

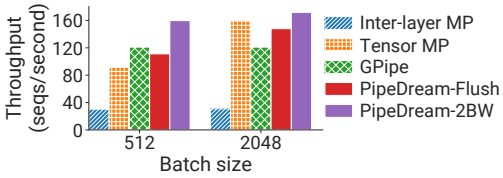
We also ran PipeDream-2BW on two BERT models: one with 2.2 billion parameters, and another with 3.8 billion parameters. Figure 10 compares PipeDream-2BW's throughput against the same baselines as before, and Figure 11 compares PipeDream-2BW's memory footprint for these BERT models. We see that results are similar to GPT. One point of difference is that GPipe does not run out of memory at the batch size of 64 (for GPT, only a batch size of 32 fits in memory, leading to a larger pipeline bubble); however, GPipe still has higher memory footprint compared to all other baselines.

### B.2. Impact of Activation Recomputation

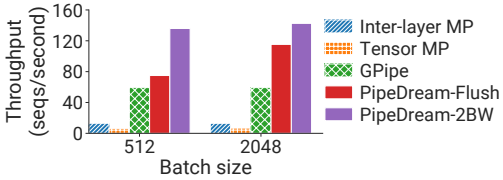
Figure 12 shows the effect of activation recomputation on throughput for various GPT models. For a given per-




**(a)** BERT, 2.2B, 8-way model parallelism (8×V100s).




**(b)** BERT, 2.2B, 8-way model parallelism (64×V100s).




**(c)** BERT, 3.8B, 16-way model parallelism (64×V100s).

*Figure 10.* Throughput of various systems for different batch sizes for BERT models. Results are shown with a single 8×V100 server, and with eight 8×V100 servers (with 16GB).

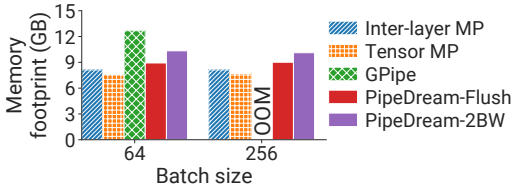
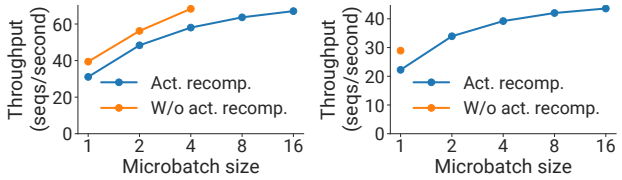



*Figure 11.* Worst-case memory footprint (in GB) of various systems with 8 V100 GPUs for a BERT model with 2.2B parameters.




**(a)** GPT, 1.3B. **(b)** GPT, 2.2B.

*Figure 12.* Throughput of $(1, 8)$ PipeDream-`2BW` configurations vs. per-GPU microbatch size for GPT models using a maximum sequence length of 512 and 8 16-GB-V100 GPUs, with and without activation recomputation. Activation recomputation helps increase the maximum per-GPU microbatch size that fits, especially for larger models, leading to higher throughput in some cases.

GPU microbatch size, recomputation introduces overhead (capped at 33% since the backward pass takes twice as long as the forward pass for most operators). However, recomputation allows for a larger per-GPU microbatch to fit on the worker, sometimes leading to higher throughput than without activation recomputation: activation recomputation leads to higher throughput in Figure 12b, but not in Figure 12a. In the extreme case (not pictured), recomputation makes it possible to train large models by reducing peak memory footprint of training, at the cost of extra compute operations due to an extra forward pass.